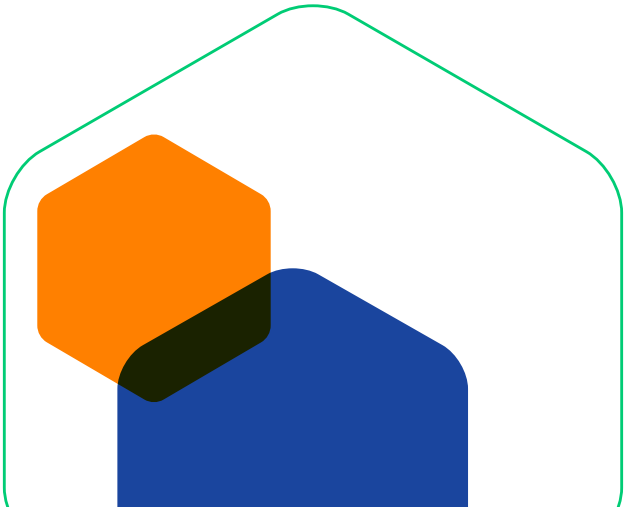
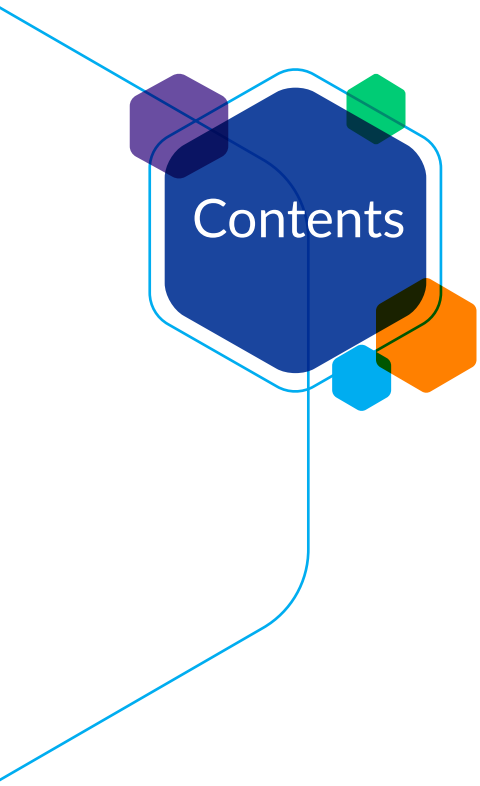
CloudBees.

A Developer's Guide to Feature Flags





Introduction

Feature flags is a term that is popping up a lot lately. You've likely heard about their benefits in de-risking and speeding up releases and how they decouple code deployments from feature release. Perhaps you've even heard how they can help get critical feedback from end users before a release so you know a feature is good to go at launch.

But, dig a little deeper, and you may realize there are many variables and considerations when using feature flags. Where do the flags go? How do you get rid of them when you're finished? How can you use them as effectively as possible to manage deployments and change management?

Whether you're brand new to the world of feature flags or have been using them for a while and want tips for managing them better, this eBook is for you!

A Refresher on Feature Flags

In its simplest definition, a feature flag (also known as a flipper or toggle) is an "*if/then*" statement in your code used to change the behavior of an application *in runtime without redeploying*.

An example feature flag in code

Feature flag deployment is gaining popularity as a way to provide safer and more effective deployments for teams looking to streamline their deployment pipeline. Feature flags simplify the process of making more frequent deployments by allowing granular control of the functionality deployed based on the environment, effectively allowing for dynamic and easy configuration of software. Some of the perks of being able to do this include:

- **Experimentation:** The ability to turn on certain features for a subset of users to determine the reception of new features
- **Safer deployments:** The ability to turn off the effects of a deployment quickly, in case a rollback of functionality is required

Further, developers love them even more because they enable trunk-based development, where they can work on disabled feature branches within the trunk. This scenario eliminates the all too common "merge hell" that commonly occurs when multiple features rolled into one release.

Feature Flags Are a Varied Bunch

As discussed in "Feature Flipper (Or Toggle): What Is It and Why Use It?" feature flags aren't one size fits all.

The simplest feature flag is a boolean value, "if and else" statements that create a conditional branch into code and allows developers to restrict some functionality to certain groups of users. That's how they started and where they get their name from.

But they can be far more complex: a function, i.e. a toggle router, that uses several inputs to determine a result. One example is a function that checks several configuration settings and even runtime data about the current session and user.

Uses for Simple Feature Toggles

The simplest feature flags control the availability of a feature for everyone. The variable is the environment in which the code runs: test, staging and production. Let's look at some examples:

- A release toggle to turn code off in production, on in staging and both off and on in development/testing (to keep testing the current code *and* test the new feature)
- An operational toggle that serves as a kill switch to allow operators to kill troublesome code or a resource-heavy feature in a graceful degradation to keep serving customers under heavy load. For example, killing a resource-heavy auto-suggestion list when a site experiences a surge in traffic

Uses for Complex Feature Flags

More complex feature flags are those that require more than a single system-wide value. At the very minimum, they use a simple boolean that varies per user. At the other end of the spectrum are flags that use multiple attributes, including non-boolean values, that vary per user *and/or* even per action taken by a user (i.e. per request).

These more complex feature flags facilitate:

- **Experiments** to A/B test a new feature
- **Canary releases** to gradually release a feature to more
- **Targeted rollouts** to get early feedback from internal staff and specific customers
- **Permissions** to control access to prized (and more highly priced) features

Characteristics of Every Feature Flag

Each feature flag has different characteristics, depending on what it is intended to accomplish:

- **Longevity:** How long you'll have it around
- **Dynamism:** When and how often you'll change its value
- **Complexity:** What data goes into determining its value
- **Authorization:** Who may change that data

We'll talk more about these characteristics and their impact in the lifecycle section next.

The Feature Flag Lifecycle

By nature, feature flags introduce complexity into a software development lifecycle and have inherent risks for teams not managing said complexity appropriately. The main culprit of this comes in the flag debt, or an overabundance of feature flags, which causes cognitive overload. This issue in turn creates further technical debt and increases potential risk in further deployments. Next, we will explain the general lifecycle of a feature flag - from creation all the way through retirement - so your team can reap the full benefit of using feature flags.

Step 1: Creating the Feature Flag

The first step when considering new functionality is to determine the type of feature flag that will be created alongside the deployment. There are a number of general categories when considering the creation of a feature flag:

- **Release toggle:** Useful for teams using continuous delivery. Release toggles essentially keep new functionality hidden in deployments while allowing for changes to be committed to the main branch. These toggles typically are kept off until functionality is ready for release
- **Experimental toggle:** Allows for turning on and off completed functionality specific to environments. This is useful when using A/B testing or canary releases. For example, in a load-balanced system, you may choose to use two workflows and randomly assign users to each, then collect data from their experiences and determine the best way forward
- **Ops toggle:** Serves as a "circuit breaker" of functionality to accommodate for potential infrastructure issues. For example, if an external service has unplanned downtime, this might be a long-lasting feature flag that gracefully deactivates functionality in your software
- **Permissioning toggle:** Allows for turning functionality on and off based on a particular subset of users. Unlike experimental toggles, Permissioning toggles are based on a specific set of users and are usually meant to be long lasting

Returning again to the characteristics of flags:

- The longevity of the feature flag and how long it is expected to be in use
- The dynamism or ease of configuring or changing the feature flag

Summarized in a table, we have:

Category	Longevity	Dynamism
Release	Days to weeks (until the functionality is deployed)	Changes with deployment or runtime configuration (turn off in case of rollback requirement)
Experiment	Weeks to months (until the data is sufficiently recorded)	Changes dynamically
Ops	Months to years (until it is no longer needed)	Changes with runtime configuration or dynamically
Permissioning	Indefinite (usually long-lasting)	Changes dynamically

The table above can help determine how a feature flag will be used when deployed into production. For instance, general functionality that will stay static when deployed will be a release flag, whereas functionality to "short-circuit" features will be an ops feature flag, changing the overall implementation.

Where a feature flag should be stored also depends on the factors above; not all storage is created equal! The table below mirrors our recommendations for where flags should live based on their longevity and dynamism, across databases, configuration files or distributed key value stores. This guidance is for those of you not using a third party feature flag management tool, of course.

	Dynamism				
Longevity	Deployment	Runtime reconfiguration	Fixed user cohort[1]	Dynamic user cohort[2]	Per request[3]
Days to weeks	configuration files, environment variables, command-line parameters	configuration files	configuration files[4]	database	
Weeks to months	X	configuration files	configuration files[5]	database	distributed key-value store
Months to years	X	configuration files	configuration files[5]	database	distributed key-value store

[1] *Used with canary releases.*

[2] *Used with pricing plans and champagne brunches.*

[3] *Used when authorization for actions is stringent or can change suddenly.*

[4] *This can be as simple as a selected range of user IDs or geographic locations given access to a feature during a canary release.*

[5] *Longer-lived toggles with a fixed user cohort are features that, for example, are only available to users in specific countries or some other demographic value. The toggle's value would be that demographic, but it would play out on a per-user basis.*

Once the feature flag is created, the next step is getting it into place—which leads us to the next section of deployment.

Step 2: Deploying the Feature Flag to Production

After creation of the feature flag, the next step is getting the feature flag into upstream environments as quickly as possible. By getting the feature flag in place, you're able to control the environment dynamically (even with the functionality being incomplete).

No matter what type of flag is in place, getting the flag set up in all upstream environments is key to the flag's functionality for the granular control it provides. Remember that the feature flag and the functionality do not need to be bundled together. For example, if there is functionality in place that may take weeks to get to a working state, you would be wise to spread out the complexity of the deployment by implementing the feature flag as soon as possible.

Once the flag is in place, the next step is using it.

Step 3: Activating the Flag

The third step in implementation of a feature flag is activating functionality in the production environment. This is where feature flags shine! Now, instead of having a deployment and hoping everything works well, you just turn the flag on when you're ready to start using said functionality.

In Step 1, Creation of the Flag, we talked about the dynamism of flags and the ability to turn them on and off. It's important, especially for ops and permissioning flags, to be able to quickly change a flag's state with ease. A tool like CloudBees Feature Management can make changing flags easy, while providing a dashboard to manage them holistically.

Finally, let's address potential scenarios you might find yourself in when considering different testing strategies:

1. One in which a team uses a standard testing environment (for example, development → staging → production)
2. Another in which a team has minimal environments and relies on continuous delivery in a single production environment, while performing testing in production

A team with multiple environments for testing can set feature flags based on the environment, enabling feature flag functionality across different environments. For instance, a deployment could deploy to both staging and production environments, then clone production data into the staging environment. Setting the feature flags in the staging environment to test new functionality can ensure functionality with real-time data.

Because feature flags are useful, many of them will reach a point where they are no longer required. Once that's the case, we'll move on to the final step of the feature flag lifecycle.

Step 4: Retiring the Flag

The final (and very important) step in the use of a feature flag is the retirement of it, effectively merging it into the standard codebase. We touched on this a bit in the introduction but want to reiterate here: feature flags have a lifespan and it is important to prune them after they are no longer required.

When using feature flags as a way to configure environments, too many feature flags will cause issues because there will be too many ways to configure an environment. In this scenario, each feature flag provides an opportunity for misconfiguration. Having too many in place causes a lot of cognitive load for those running configuration, which eventually causes issues down the road.

The solution to this? Create a process for determining when feature flags are regularly retired and removed from the codebase. This system will result in deployments that simply remove feature flags, causing certain features to become standard in the codebase. Release flags will usually be retired quickly after a successful release. Experimental flags will be retired after successful data is collected and ops and permissioning flags may end up sticking around for the long term.

Best Practices for Management of Flags

Now that you have a better understanding of how complex feature flags can be, along with the typical lifecycle of a flag, let's look at our best practices for managing flags. This guidance will ensure you can properly scale flag usage out across teams and keep all of your developers from being at each other's throats.

Create a Standard Org-wide Naming Convention

A proper naming convention for feature flags is the first step towards better management. In the absence of a naming convention, it's possible for team members to confuse the flags or worse, create flags with the same name. If you don't label your flags carefully, someone in your team could mistakenly trigger these flags, which can lead to severe disruptions in extreme cases. Adopt a naming convention that allows you to get information about the type or the purpose of the changes introduced to a flag. CloudBees Feature Management provides you with a unique approach that enforces name uniqueness under namespaces based on the type system of the programming language you are using.

Create a System for Logging Changes

In order to track changes better, consider logging all the changes to a feature flag. If you set up logging, you can track who introduced a change and at what time. In distributed teams, this can save numerous hours in figuring out who is responsible or has the rights for a particular change. The effects of feature distribution are not restricted to the engineering team- you should log the feature status to your usage analytics, support systems and more. With CloudBees Feature Management, we have exposed the impression handler to facilitate reporting to all your different platforms.

Include the Flag Type in the Label

As we've discussed, feature flags can be classified into four different types: release, operations, experimental and permissioning flags. Classifying the four types of flags in different configuration files can help you reduce complexity and streamline support and debugging when errors occur. Build a label system into your management. (Tools such as ours have this labeling functionality built in.)

Ensure There is Proper Access Control

Putting feature flags in the hands of non-technical users can be a double-edged sword. (No offense, product managers!) Use a granular approach to access controls. Different teams should have different levels of visibility and access to environments and flags. For example, you can consider imposing restrictions on temporary logins for those with support credentials.

Create a Standard Process for Flag Removal

Over time, feature flags can contribute to code complexity with deprecated branches and features that were never released or were replaced with something else. Current flags can conflict with previous ones. It can be a challenge to identify which flags are required and which ones are redundant or obsolete. One way to figure this out is to see if a feature flag is always on or off; if so, it has served its purpose. You can also define the flag status to help your teams distinguish between short-term and long-term flags. The status field can be dynamically updated after a defined period, allowing your team to identify which flags are safe to remove.

To avoid technical debt from building up, you need to carefully manage flags with precise control and visibility into their changes, rollouts and sunsetting. However, with bootstrapped, homegrown systems, all this can be challenging. Using an enterprise feature flag tool will help to simplify and even automate this process.

Recommendations for Managing Deployments Using Feature Flags

Ok, you now understand the different types of feature flags and how complex they can be, their typical lifecycle in the organization and how to manage them across teams. Now the fun part! Let's look at all the different ways you can use feature flags in a deployment.

Selective Rollout

You can use a selective rollout for enabling certain features for premium users or beta users. Many products offer paid features that can only be accessed by the users that pay for the premium functionality. You can develop your own in-house plugin for managing premium features or enterprise feature management platforms offer the same functionality built in. They provide ready-made plugins and code snippets via their SDK to quickly implement different types of feature flag use cases.

Geographical Feature Flag Rollout

Secondly, you can use feature flags to roll out features for only specific geographic regions or countries. Imagine you own a financial product that isn't authorized to operate in certain countries. To comply with local regulations, you need to disable certain features for users in a specific country or region. Feature flags can help you accomplish this task.

Roll Out Faster to Paid Users

The concept of rolling out new features faster to paid users isn't that common. However, it's similar to how Patreon works. Patreon allows creators to receive contributions for the content they create. In return, contributors can access new content earlier than non-contributors. The concept of rolling out new features faster to paying users works for certain use cases such as content creation.

Advanced Feature Flag Activation

What if you want to enable feature X on your staging environment so your product manager can test the feature but disable the feature for your production environment? With a custom work on your homegrown system or using an enterprise management platform, you can create custom configurations based on specific branches or other configuration items that determine the activation or deactivation of features.

Percentage Rolling Releases Based on User Feedback

Now let's get to the really fun stuff, where we explore a complex use case. In this scenario, we want to roll out a new feature gradually to a growing percentage of our user base until we have released the feature to all users. For example, every day, we release a new feature to five percent of the user base.

By slowly rolling out the feature, you collect quantitative user feedback. For example, you can ask users to rate the feature on a scale of 1 to 10. Next, you implement a snippet of logic that reverts a new feature if 30 percent of the users report a score below five out of ten. Of course, you can play with the numbers. Ultimately we are looking at automating a feature rollout based on live user feedback, which should be a product manager's dream!

Consider the Factors for Building In-House vs. Buying

We hope this guide has been useful for helping to get your feature flag program up and running or for getting your current program running more efficiently. By now, you probably realize that properly scaling out your management of flags will take additional resources for coding and maintenance over time. We've found that the wide majority of our current customers start out developing their own system, and eventually realize that they can save time and money by using an enterprise-ready third party platform. This journey is different for every company, but here are some questions to consider as you grow your own program and consider the same decision:

1. Do I have the resources to spare from my core business to create a flag system?
2. How much time and money am I willing to spend on maintenance of the system?
3. What hardware and software investment do I need to create the system?
4. Will I be able to scale my system not only for other development teams but also PMs?
5. Will the flag system be compatible with my current production pipeline?
6. What is the long-term plan for the system?

If you are looking for more information, we have a separate guide "Feature Flag Management: Should I Build or Buy" that takes a deep dive into these questions. It also covers enterprise needs around dashboards, permissions, analytics and reliability. Good luck!

Learn More

- **Watch the Webinar**
  Growing DevOps Maturity Using Feature Flags
- **Read the Guide**
  The Ultimate Feature Flag Guide
- **Review the eBook**
  Feature Flags: Should I Build or Buy?

0122v00

CloudBees, Inc.
4 North Second Street | Suite 1270
San Jose, CA 95113
United States
www.cloudbees.com
info@cloudbees.com